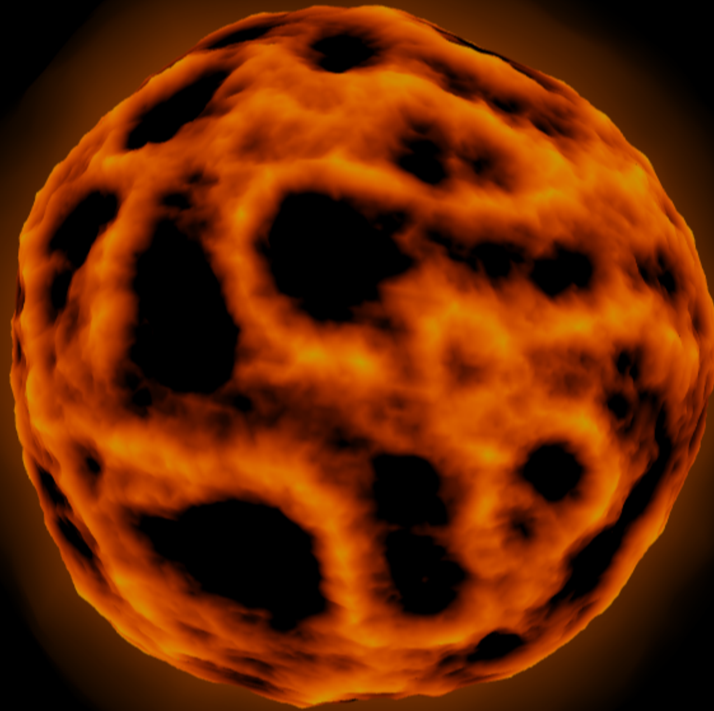
Solar system with editable procedural sun

TNM098 - Procedural Methods for Images

Aniisa Bihi

May 2020



LINKÖPINGS UNIVERSITET

Abstract

This report covers the project process of creating a solar system with a editable procedural sun as the main object. The report covers subjects and methods that have been used in the project. The main frameworks and libraries used are *WebGL*, *JavaScript*, *Three.js*, *OrbitControls.js* and *dat.GUI*. Conclusions and methods in the report are based on documented theories and the authors own speculations. The result of the project is a web-based application that contains a solar system with a procedural sun and planets with textures. The parameters set for the sun can be changed and experimented with using controls provided by a GUI.

The final project can be viewed at:
aniisabihi.github.io/Procedural-Solar-System

Contents

1 Background **1**
- 1.1 Procedural Noise Functions . . . .
- 1.2 WebGL . . . .
- 1.3 Three.js . . . .
  - 1.3.1 OrbitControls.js . . . .
- 1.4 Dat.GUI . . . .

2 Method **4**
- 2.1 Scenery . . . .
  - 2.1.1 Stars . . . .
  - 2.1.2 Planets . . . .
- 2.2 The Sun . . . .
  - 2.2.1 The Sun's Glow . . . .
  - 2.2.2 Noise . . . .
  - 2.2.3 Vertex Shader . . . .
  - 2.2.4 Fragment Shader . . . .

3 Result **10**
- 3.1 The Solar System . . . .
- 3.2 The Procedural Sun . . . .
- 3.3 The GUI . . . .

4 Reflection **15**

Listings

1 Generate planet function . . . .
2 Creating the sun . . . .
3 The sun's vertex shader . . . .
4 The sun's fragment shader . . . .

List of Figures

1 Structure of a Three.js application
2 The view of a perspective camera and its properties
3 Perlin noise pattern
4 Final application
5 Solar system zoomed out - View from the side
6 Solar system zoomed out
7 The sun
8 The sun with maximum displacement
9 The sun with no displacement
10 The sun with colours changed
11 The colours applied to the sun with the GUI
12 Application's GUI
13 Close image of the sun
14 Worley noise pattern

1 Background

The general idea of this project was to create a solar system with one main planet created with *procedural methods*, and also editable through a *GUI*. Since the center of our solar system is the sun, it was chosen as the main planet. The aim was that focus would be on the sun, created through imagination, while the other planets would be generated in a simpler fashion. With a main editable planet, it was decided that the final interface would be hosted through *GitHub* with the use of *WebGL*, *Three.js*, *OrbitControls.js* and *dat.GUI*.

1.1 Procedural Noise Functions

Procedural programming is a form of structured programming that is based on the concept of calling procedures. These procedures consist of series of computational steps to be carried out. When a program is executing, procedures can be called at any point during the execution [1]. Procedural procedures are also known as procedural functions, with procedural noise functions being widely used in *Computer Graphics*. Procedural noise functions is a way to add complex and intricate details at low memory and authoring cost [2].

1.2 WebGL

Web Graphics Library (WebGL) is a *JavaScript API* used to render interactive 2D and 3D graphics within the web browser. WebGL can, based on code from functions, draw points, lines and triangles. These points, lines and triangles can together draw more complicated geometries if necessary functions are provided. WebGL runs on computers *GPU* and to make that possible, a vertex shader and a fragment shader are needed. These shaders are written in *GL Shader Language* (GLSL), which is a strictly typed *C/C++* language. Pairing a vertex shader with a fragment shader will result in a shader program. The vertex shader computes vertex position outputs which lets WebGL rasterize primitives like points, lines or triangles. When these primitives are being rasterized, the fragment shader computes a colour for each pixel of the primitives that are being drawn. Data can be provided to the shaders through buffers, attributes, uniforms, textures and varyings. Buffers are arrays of binary data that usually contains normals, positions, texture coordinates or vertex colours. Attributes are then used to specify how to pull data from buffers and provide them to the vertex shader. Uniforms are global variables that are defined outside of the shader program. Textures are arrays of data the usually contains image data but can also contain something other than colours. Lastly, varyings are used to pass data from the vertex shader to the fragment shader [3].

1.3 Three.js

Three.js is a JavaScript framework used to create and display animated 3D Computer Graphics on the web. Three.js uses WebGL and is hosted in a repos-

itory on GitHub [4]. To draw advanced objects with WebGL requires a lot of code, and that is why Three.js can be useful. It handles things like scenes, lights, shadows, materials, textures, 3D, maths and other things that would require a lot of code if written with WebGL directly. A Three.js app contains objects that are connected to each other and its structure is fairly easy to understand. The following figure shows a simplified diagram structure of a Three.js application [5].

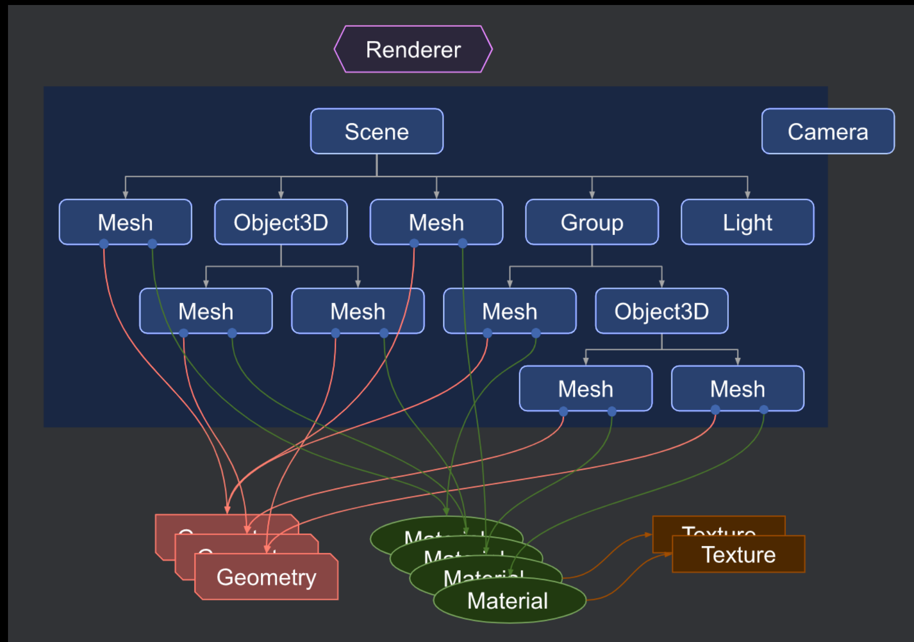


Figure 1: **Structure of a Three.js application**

The *renderer* is the main object of Three.js, a *scene* and a *camera* are passed to the renderer to draw the part of the 3D scene that is inside the *frustum*, see Figure 2. In Figure 1 we can see that the camera is halfway out of the scene graph, this represents the fact that a camera does not have to be in the scene graph to function. There can, for example, be multiple camera objects in a scene graph. This is because the camera will move and orient relative to its parent object. A scene object is the root of a scene graph and is the parent to objects like *mesh*, *object3D*, *light* and *group*. A mesh object needs a *geometry* and a *material* to be drawn. Geometry objects contain vertex data of geometry like a sphere, cube, plane, etc. With Three.js, these geometry objects are built in as geometry primitives and therefore only requires simple function calls to be created. This is also possible with material objects that contain surface properties, for example colour. A material can also reference *texture* objects that contain images loaded from files, generated from a canvas or rendered from

another scene [5].

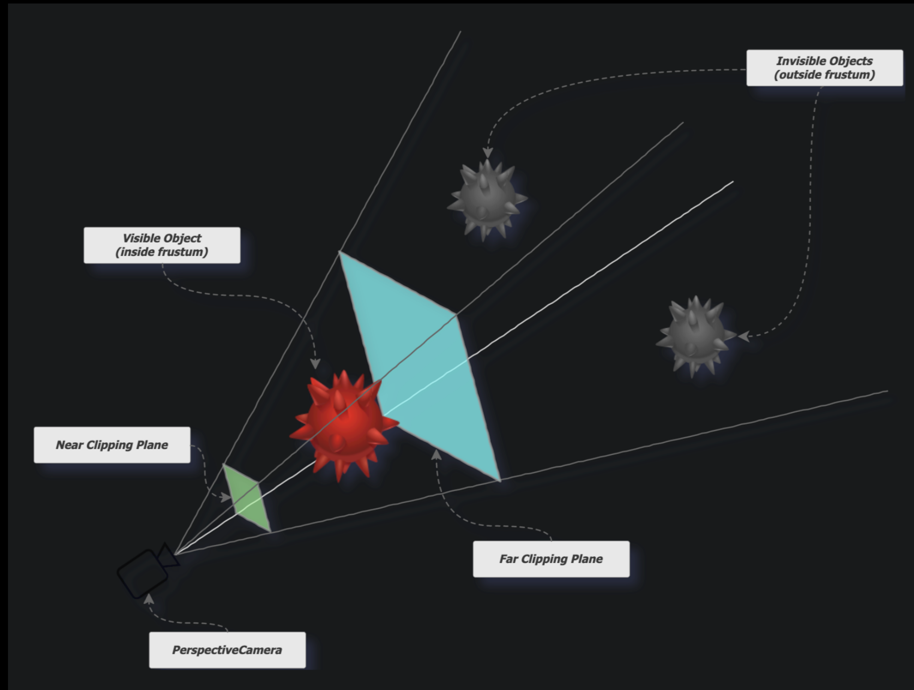


Figure 2: **The view of a perspective camera and its properties**

1.3.1 OrbitControls.js

OrbitControls.js is a script that is included in the Three.js repository on GitHub. With the use of orbit controls the camera can manually (or automatically) orbit around a target. The orbit controls can be modified to let the user manually move the camera around the target with either their mouse or keyboard. With the ability to move the camera manually, the user can navigate the scene and view it from different perspectives [6].

1.4 Dat.GUI

Dat.GUI is a lightweight controller library for JavaScript that works as a graphical user interface. The library does not require many lines of code to create an interface that can modify JavaScript variables with direct results. This is all possible without having to redeploy code [7]. All necessary code to use the library is available at Github. The easiest way to use the library is by using the built source in the GitHub repository called dat.gui.min.js. The built file bundles all the necessary dependencies to run dat.GUI [7].

2 Method

2.1 Scenery

The scenery in this project includes everything in the scene except the sun, since it is our main object. The scenery consist of stars, the solar systems planets and their orbits.

2.1.1 Stars

In the final scene there are 4 million stars placed at a random position. Creating the individual stars at a random x position required giving each star their own vector with the same x-, y- and z-position. The stars were created using point geometry and a star texture. The texture was manipulated as a mapped material with transparency and additive blending. All the stars together form a particle system that is continuously rotated at a negative y-axis.

2.1.2 Planets

The function for generating planets is shown in Listing 1. It starts by creating a sphere geometry with a given radius and also loading in the given texture for the planet. This texture is then wrapped around the sphere geometry in a way that makes the texture match a sphere. Then the texture is added to a Phong material and also made double sided. The geometry and material are then meshed together to create the planet object. After creating the planet some orbit values are set; radius, rotational speed, orbit speed and position. Lastly, the planet is added to the scene. A separate function creates the planet's orbit line and the planet's movement along the orbit line is updated in the render function.

The Earth is also created in a similar fashion though it does not use the described function. This is because the Earth also has a bump map on its material to make the land levels higher than the ocean levels. The Moon however, was created using the functin described in Listing 1 and then linked with the Earth object and joining the Earth group. The Earth group also consists of clouds as a separate object and the Moon orbiting around the Earth.

```
// Function that generates planets with images as textures
function generatePlanet(radius, tex, radX, radY, radZ, rotSpeed,
    orbitSpeed)
{
  var geometry = new THREE.SphereGeometry( radius, 32, 32 ),
    texture = new THREE.TextureLoader().load( tex );

  //Wrap texture
  texture.wrapS = texture.wrapT = THREE.MirroredRepeatWrapping ;
  texture.repeat.set( 1, 1 );

  var material = new THREE.MeshPhongMaterial({
      map : texture,
      side : THREE.DoubleSide
    }),
    planet = new THREE.Mesh( geometry, material );

  // Orbit radius and rotations
  planet.orbitRadiusX = radX;
  planet.orbitRadiusY = radY;
  planet.orbitRadiusZ = radZ;
  planet.rotSpeed = rotSpeed;
  planet.rot = 2;
  planet.orbitSpeed = orbitSpeed;
  planet.orbit = Math.PI * 2;
  planet.position.set(radX, radY, radZ);

  scene.add(planet);
  return planet;
}
```

Listing 1: **Generate planet function**

2.2 The Sun

The sun was created with Icosahedron Geometry using a higher level of detail to give it a spherical appearance. The material used is a shader material that takes in uniforms that include the sun's colour and noise values. The material also includes a vertex shader and a fragment shader, that are both presented in 2.2.3 and 2.2.4 respectively. The following subsections will present more details of the sun as it was created using imagination.

```
// Adding the sun to the scene
var sunSize = 20,
  sunGeometry = new THREE.IcosahedronGeometry( sunSize, 6 ),
  sunMaterial = new THREE.ShaderMaterial({
        uniforms: {
          highlight: {value: new THREE.Color(highlight)},
          highlightBlend: {value: new THREE.Color(highlightBlend)},
          blend: {value: new THREE.Color(blend)},
          innerGlow: {value: new THREE.Color(innerGlow)},
          displacementSize: {value: noiseDisplacement},
          noise_size: {value: noiseSize},
          time: {value: 0}
        },
        vertexShader:
            document.getElementById('vertexShader').textContent,
        fragmentShader:
            document.getElementById('fragmentShader').textContent,
        side: THREE.DoubleSide
    });
var sun = new THREE.Mesh( sunGeometry, sunMaterial );
scene.add( sun );
```

Listing 2: **Creating the sun**

2.2.1 The Sun's Glow

The glow of the sun works as a separate object, created the same way as the sun. The glow uses the same noise functions with similar fragment and vertex shaders, explained further in 2.2.3 & 2.2.4, with just an added aspect of intensity and opacity. The intensity and opacity can be adjusted using the GUI. In short, the sun's glow is not a part of the sun object but shares uniforms defined outside of respective shader program. These uniforms are the noise's size, displacement and time. Therefore, if the noise uniforms are changed using the GUI, it will affect both the sun and the sun's glow. But changing the sun's colour will not affect the glow's colour, as they are not connected.

2.2.2 Noise

Procedural noise has been used in this project to create the appearance of the sun. This includes the sun object's displacement, the blending of the sun's colours and the movement of the sun. The noise function used in this project is Perlin Noise.

2.2.2.1 Perlin Noise

Since Perlin noise has a pseudo-random appearance, with its visual details the same size, it was chosen as the procedural noise function for this project. It

is also common to use Perlin noise for creating computer-generated fire, which fit the purpose as the sun can be seen as big fire ball. Creating a fire ball was exactly the imagination in works during this project, and Perlin noise helped capture that controlled random appearance. More precisely, the Perlin noise functions used can be found at github.com/ashima/webgl-noise.

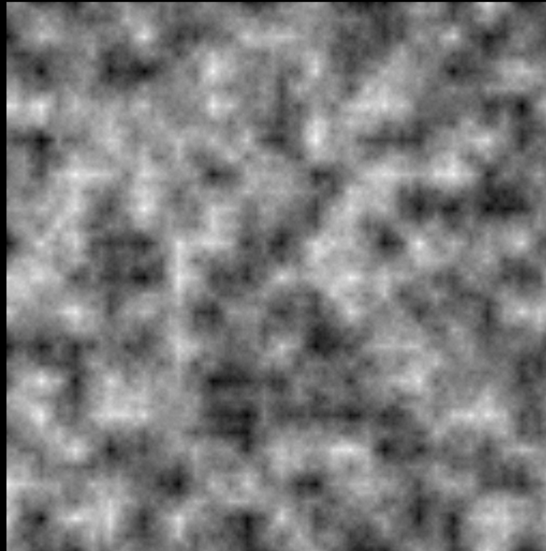


Figure 3: **Perlin noise pattern**

2.2.3 Vertex Shader

The vertex shader shown in Listing 3 also contains the Perlin noise functions described in 2.2.2.1. In this shader we disturb the vertex position of each vertex, along the direction of its normal, to get the sun's shape to change depending on the time value. This is being done using the turbulence function. Since we are aiming for a random but controlled appearance, the Perlin function is used inside the turbulence function. There is also an additional distortion, b, to help disturb the sphere shape even further. A lot of numbers and factors in the vertex shaders were experimented with, tweaking the numbers until the desired result was reached.

Lastly, the displacement works by having a displacement factor that multiplies with the displacement created with noise, and there after adding the normal multiplied with the displacement to the original position. The displacement factor is referred to as "displacementSize" and is a GUI parameter. The GUI parameters are uniforms of the vertex shader that can alter the sun's noise as the values of the GUI are being changed.

```
varying float noise;
varying float noise2;
uniform float time;

// Gui parameters
uniform float displacementSize;
uniform float noise_size;

float turbulence( vec3 p ) {
  float t = -.35;
  for (float f = 1.0 ; f <= 10.0 ; f++ ){
    float power = pow( 2.0, f );
    t += abs( pnoise( vec3( power * p ),
        vec3( 10.0, 10.0, 10.0 ) ) / power );
  }
  return t;
}

void main() {

  noise = 18.0 *  noise_size * turbulence( normal + time );
  noise2 = 10.0 *  noise_size * turbulence( normal + time );
  float b = 5.0 * pnoise( 0.05 * position + vec3( 2.0 * time ),
            vec3( 100.0 ) );
  float displacement = - 5. * noise + b;

  vec3 newPosition = position + normal *
                    ( displacement * displacementSize );
  gl_Position = projectionMatrix * modelViewMatrix *
                vec4( newPosition, 1.0 );
}
```

Listing 3: **The sun's vertex shader**

2.2.4 Fragment Shader

The fragment shader is a lot shorter than the vertex shader and consists of mixing the colours and the noise together to create fragment colour. The shader takes in the noise floats created in the vertex shader and also the uniforms specified in Listing 2. It is in this shader that the appearance of the sun really takes place and the pattern we can seen in Figure 7 takes form. The GUI parameters are uniforms of the fragment shader that can alter the sun's appearance as the values of the GUI are being changed.

```
varying float noise;
varying float noise2;

// Gui parameters
uniform vec3 highlight;
uniform vec3 highlightBlend;
uniform vec3 blend;
uniform vec3 innerGlow;

void main()
{
  vec3 colormix1 = mix(highlight, highlightBlend, (1. - 2. * noise2));
  vec3 colormix2 = mix(blend, innerGlow, (1. - 2. * noise));
  vec3 finalColor = mix(colormix1, colormix2, (1. -2. * noise));
  gl_FragColor = vec4( finalColor, 1.0);
}
```

Listing 4: **The sun's fragment shader**

3 Result

The final application with the solar system and the GUI can be seen in Figure 4 and viewed at aniisabihi.github.io/Procedural-Solar-System.

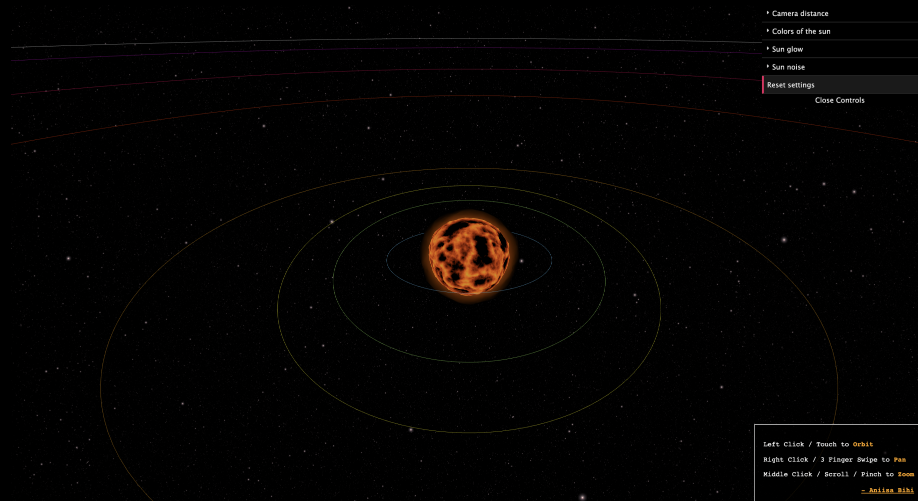


***Figure 4:* Final application**

3.1 The Solar System

The full solar system completely zoomed out can be seen in Figure 5 and Figure 6.

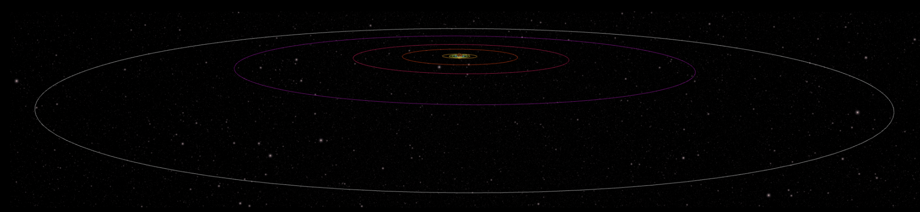


***Figure 5:* Solar system zoomed out - View from the side**

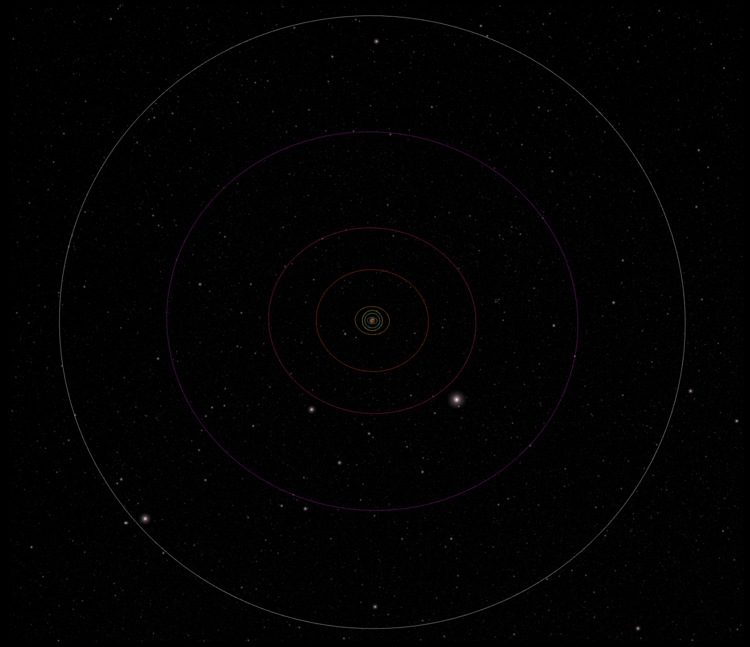


Figure 6: **Solar system zoomed out**

3.2 The Procedural Sun

The sun, illustrated on the report's front page, can be seen in Figure 7.

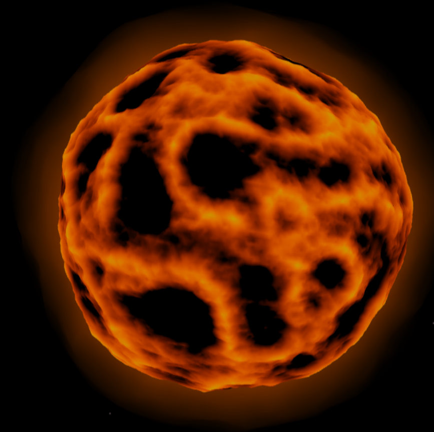


Figure 7: **The sun**

The sun with maximum and no displacement changed with the GUI can be

seen in Figure 8 and Figure 9.

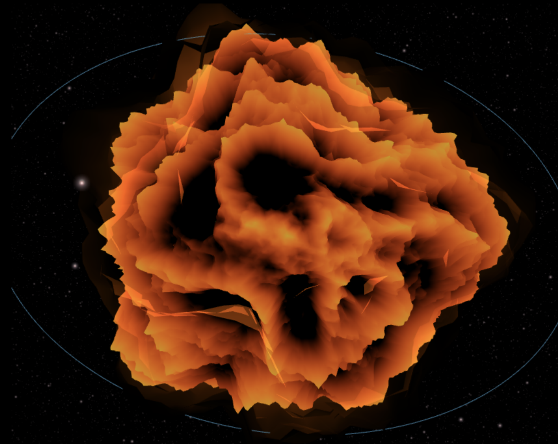


Figure 8: **The sun with maximum displacement**

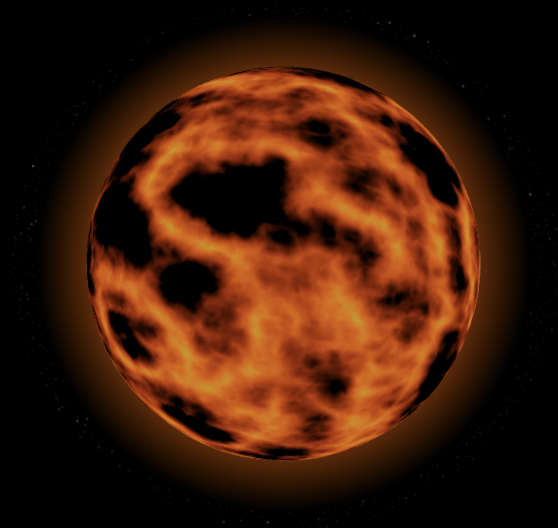


Figure 9: **The sun with no displacement**

The sun with colours changed with the GUI can be seen in Figure 10.

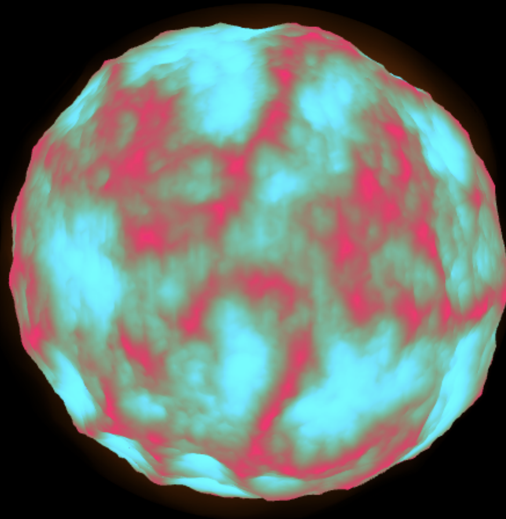


Figure 10: **The sun with colours changed**

▾ Colors of the sun

Highlight Color	0xff00be
Blending Color	0xff96
Inner Glow	0x3fe0eb

Figure 11: **The colours applied to the sun with the GUI**

3.3 The GUI

The application's GUI can be seen in Figure 12.

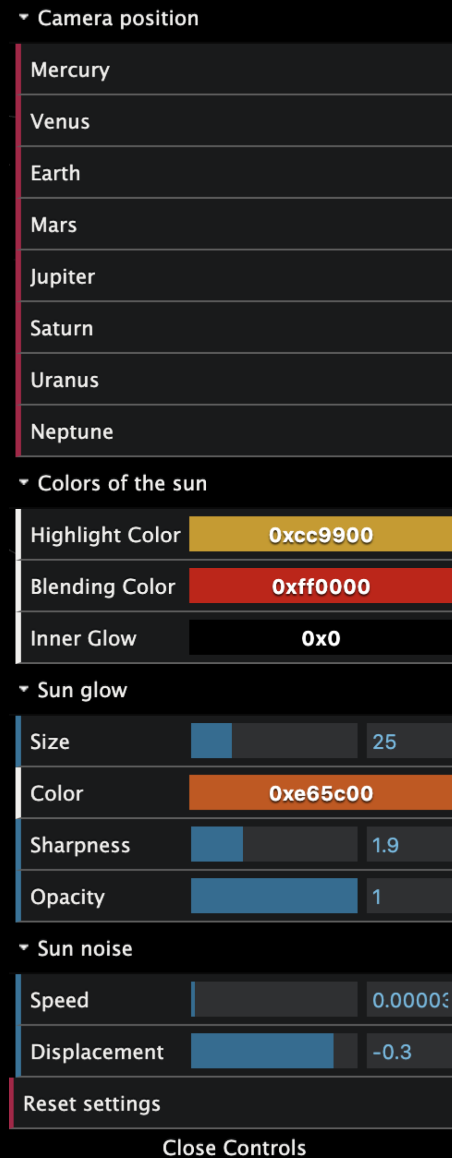


Figure 12: **Application's GUI**

4 Reflection

During this project, a lot of the implementation were done based on pure imagination and experimenting. My goal was to create a sun that looks and moves the way I imagine it. I therefore wonder if I would have produced a better result if I had a reference image and did more research. One thing that I noticed after the project's implementation was done, was that the pattern of the actual sun is similar to Worley noise. In Figure 13, we can see a close image of the sun that resembles the pattern that can be created with Worley noise, see Figure 14. Had I aimed to create a realistic sun then maybe I would have chosen Worley noise instead, which makes me wonder how the result would have changed and if it is something I want to do in future work. I also did not realise how much time it would take to set up the scenery, and therefore spent more time on that than I would have liked. This consequently lead to loosing time on experimenting more on the Sun and other methods. However I do not regret the time spent on the scenery as I see a lot of potential in the Three.js framework and look forward to using it in future WebGL projects. Working out of my imagination (in combination with some realism) made the project have no bounds, which made for a lot of trial and error. I am however very proud of the end result and can definitely say it has been an educational project.

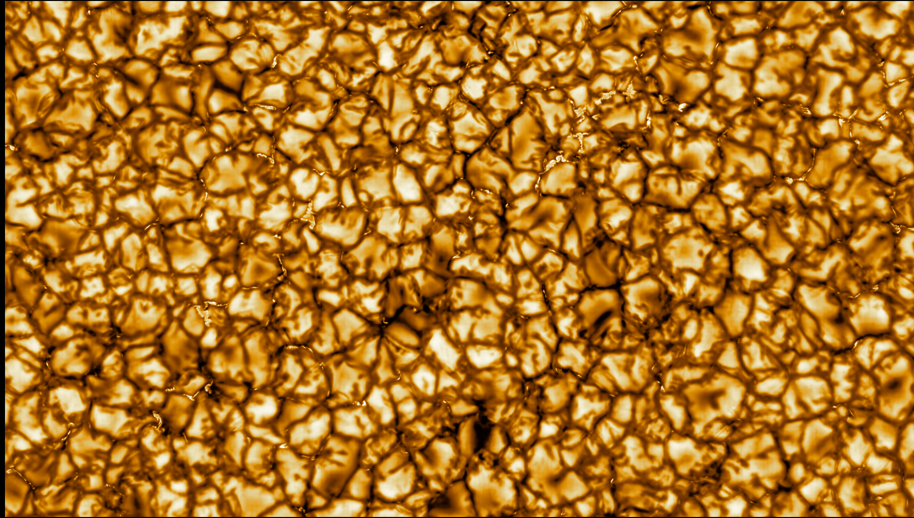


Figure 13: **Close image of the sun**

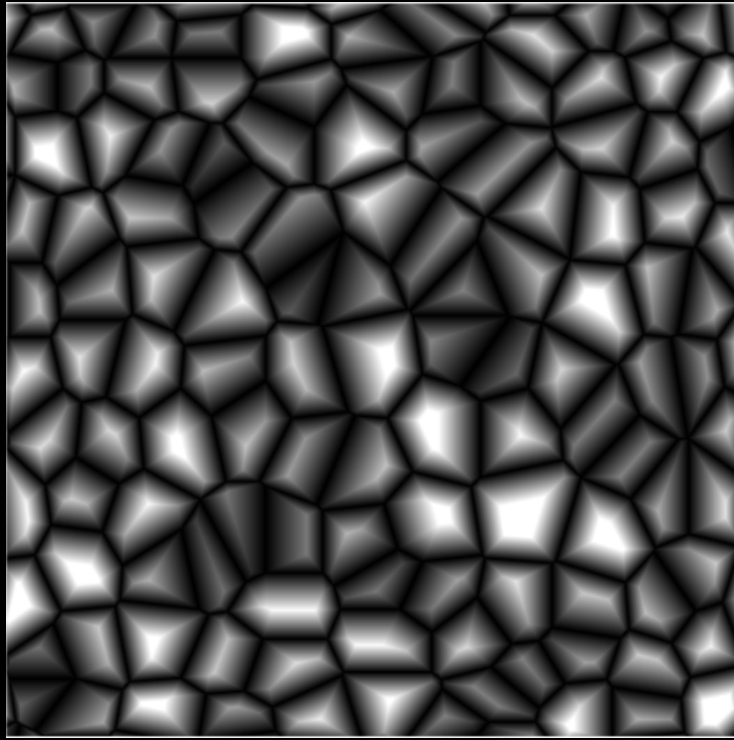


Figure 14: **Worley noise pattern**

References

[1] Pankaj Patel. *Differences between Procedural and Object Oriented Programming*. URL: https://www.geeksforgeeks.org/differences-between-procedural-and-object-oriented-programming/. (accessed: 21.05.2020).

[2] Ares Lagae et al. "State of the Art in Procedural Noise Functions". In: *EG 2010 - State of the Art Reports*. Ed. by Helwig Hauser and Erik Reinhard. Eurographics. Norrkoping, Sweden: Eurographics Association, May 2010.

[3] *WebGL Fundamentals*. URL: https://webglfundamentals.org/webgl/lessons/webgl-fundamentals.html. (accessed: 21.05.2020).

[4] Lewy Blue. *What is three.js?* URL: https://discoverthreejs.com/. (accessed: 21.05.2020).

[5] *Three.js Fundamentals*. URL: https://threejsfundamentals.org/threejs/lessons/threejs-fundamentals.html. (accessed: 21.05.2020).

[6] *OrbitControls*. URL: https://threejs.org/docs/#examples/en/controls/OrbitControls. (accessed: 21.05.2020).

[7] Google Open Source. *dat.GUI is a lightweight controller library for JavaScript*. URL: https://opensource.google/projects/datgui. (accessed: 21.05.2020).